Reid Elkins

Homework 6 - CSCE 686

June 05, 2020

Problem 1:

Talbi 1.1 - A greedy algorithm for the vehicle routing problem:

While set of nodes j is not empty:

For each van i in the fleet K:

Add node x to the path of van i where the weight of the package dropped off at x does not exceed the remaining capacity in van i and cost of path from the current node that i is at to the given node x is minimal for all nodes in j.

Remove node x from set j

http://www.optimization-online.org/DB\_FILE/2018/06/6645.pdf

Constraints which can be added to the problem.

1. If vehicles all have a subset of the nodes which they are limited to being able to visit. This could be the case if there is a collection of potential services which each vehicle could offer, but each only has a subset of the possible services and each node has a specific service which they are looking for.

2. Another potential constraint is one that is similar to ride-share systems where there is a set of varying drivers in varying places along with varying nodes which pop up at any given time.

3. Lastly, if the vehicles are not only dropping off items but picking up items, like the USPS, this would affect the trucks capacity at any given time especially if the pick up items are of unknown size.

Talbi 1.17 - The cost of a given solution for the vehicle routing problem takes into account the path taken by each vehicle, it is not solely based upon the customer which each vehicle visits. Computing the incremental optimization function of moving one customer from one vehicle to another is difficult because when a customer is switched, the new paths need to be found to optimize the cost for the two vehicles proposed in this situation which can be a computationally complex process.

Problem 2 :

**i.)** I will be solving the knapsack problem using a BF\* variation. The knapsack problem is one of the np complete problems I will be incorporating into my project in which I will be finding the best subset of stocks to buy from a given pool of stocks. For this problem, I will consider that there is a set amount of money, only 1 share of each stock can be bought, the cost of the stock will be based on the close of market price on June 5th, 2020, and the value of the stock will be arbitrarily signed for now. For my project the values will not be as arbitrary.

Given an amount of money m and a set of stocks j, each with a certain price p and value v, find the subset of stocks that maximizes the value while the sum of the price of the stocks is less than the money held.

Domains D:

Input Di: dictionary of stocks: {S: (P, V)}, S: stock ticker, P: stock price, V: value, M: money

Output Do: list of stock tickers chosen, B

Objective function:

maximize ∑zi=1 {Si:Vi} where ∑zi=1 {Si:Vi} <= M where S is in B and z is the size of B

The knapsack problem is NP-Complete:

The 3-SAT problem (NP-Complete) can be reduced to the Subset Problem in polynomial time which can easily be reduced to the 0-1 Knapsack problem by setting the cost in the subset problem to the weight of each item in the knapsack problem and the given sum is now the max capacity of the knapsack.

**ii.)**

***algorithm domain requirements speciﬁcation form:***

*• name: Global-Search Breath First Search (Di, Do); gs-bfs*

*• domains: Di - list of stock tickers, list of prices, list of values, integer money amount*

*Do - max value of stocks with sum of the correlating prices less than money*

*• operations:*

*I(x); x in Di; x is a possible candidate from input set*

*O(x,z); x in Di, z in Do; z is an optimal (maximal) solution or set of*

*optimal solutions (or satisﬁzing solutions)*

***algorithm domain design speciﬁcation form:***

*• name: Global-Search-bfs (Di, Do); gs-bfs*

*• domains: Di is set-of-candidates, Do are the sets of solutions, Dp is set of partial solutions of generated nodes*

*• imports: ADT list, dictionary, double, integer*

*• operations:*1

*I(x); x in Di*

*O(x,z); x in Di, z in Do;*

*I’(x,y); x in Di, y in Dp;*

*Dp is the “open” list;*

**–**

**–** *deﬁne state*

**– *next-state-generator***

*i)* ***selection*** *of a partial solution y in Dp based upon the highest potential total value that the solution could produce*

*ii)* ***Generation*** *of next states by adding node that includes next stock if it can be added and the node that does not include next stock*

**– *feasibility*** *(xj , y) − >* if potential cost is greater than or equal to current upper bound, and solution to Dp

**– *solution*** *(y) − > max summed value of combination of stocks where sum of the stock prices is less than or equal to the amount of money*

**–** *objective solution(Dp) − > “ordered dictionary of nodes whose cost is greater* than or equal to the current greatest upper bound”

**– heuristics**  come from problem domain insight:

Upper bound of each node calculated as:

while <= money

Cost of each node calculated as:

where if current + > m,

Add (

Keep track of the greatest upper bound found at any node to that point.

If the found cost of any node is less than the global upper bound, kill that node

***algorithm domain function speciﬁcation form: (iterative)***

*•* ***function*** *global-search-bfs-iterative (Di) sets in Do*

*• Initial condition: clear(set Dp); deﬁne initial x in Di and associated initial state s*0 *in Dp; si is search state/node.*

*Dp is the “open” list;*

*• body*

*while Di for each state not exhausted do gs-dfs-loop: “all nodes/states expanded”*

**– *next-state-generator***

i) find the node with the max cost in the dictionary of potential nodes

*ii) “****Generation****” of all next states of s(y). Add node that includes next stock from current node as long as sum of prices of included nodes does not exceed the amount of money. Then add the node that does not include the next stock in the list to the potential node dictionary*

*delete s(y) from Dp*

**–** if **feasability** (s(xj, y)), if found cost of newly generated node is less than current upper bound, add node to dictionary of potential nodes

**–** *if* ***solution(s)*** *then save z = s, z in Do, “current optimal solution”*

**–** *end gs-dfs while loop*

**Next iteration:**

*•* ***function*** *global-search-bfs-iterative (Di) sets in Do*

*• Initial condition: clear(set Dp); deﬁne initial x in Di and associated initial state s*0 *in Dp; calculate cost and upper bound of initial x, set global upper bound to find upper bound*

*Calculation formulas:*

Upper bound of each node calculated as:

while <= money

Cost of each node calculated as:

where if current + > m,

Add (

*• body*

*while Di for each state not exhausted do gs-dfs-loop: “all nodes/states expanded”*

**– *next-state-generator***

i) find the node with the max cost in the dictionary of potential nodes, if multiple potential nodes have same least cost, pick the first one that was added

*delete s(y) from Dp*

*ii) “****Generation****” of all next states of s(y). Next states: one which includes the next potential stock and one which does not include the next potential stock. Generate upper bound and potential cost value of each node*

**–** if **feasability** (s(xj, y)), if found cost of newly generated node is less than current global upper bound, add node to dictionary of potential nodes

**-** if either node that is now a potential solution had a greater upper bound than the global upper bound, look through all potential nodes in current dictionary and remove those that have a cost less than the new upper bound

**–** *if* ***solution(s)*** *then it is the last node looked at before the while loop ends*

**–** *end gs-dfs while loop*

**iii.)** Code is in github link here:

<https://github.com/reidelkins/CSCE686/tree/master/CSCE686_HW6>

**iv.)** Algorithm complexity in the worst case is still 2^n if it has to search every node. In the best case, the algorithm allows pruning of the entire tree except one path from the root to a leaf node.

**v.)** The goal of this is to become familiar with a NPC problem and be able to create a top-down algorithm for one. In order to do this, I went through the top down approach to create the algorithm and then coded it in python. I tested the algorithm on list of 4 stocks, 10, stocks, and 20 stocks.

Here are the results found

|  |  |
| --- | --- |
| Stocks Looked At | Time |
| 4 | .01s |
| 10 | .01s |
| 20 | .01s |

It seems from the time outputs gathered from the time command in the linux terminal, there was no significant time difference for the increased size of stock list that were tested by the program. Of course with list of 100 stocks or larger we would expect that the time would increase from .01s but this does show that the best first heuristic chosen in the program is pruning large amounts of the tree and the program is able to run very quickly.

References for problem 2:

<https://www.youtube.com/watch?v=yV1d-b_NeK8>

<https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/>

<https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/>

<http://cs.franklin.edu/~shaffstj/cs319/week12.htm>

<https://www.cc.gatech.edu/~rpeng/CS3510_F16/notes/Nov28knapsackNPC.pdf>

Lecture4\_gs\_bfs\_20.doc by Dr. Gary Lamont